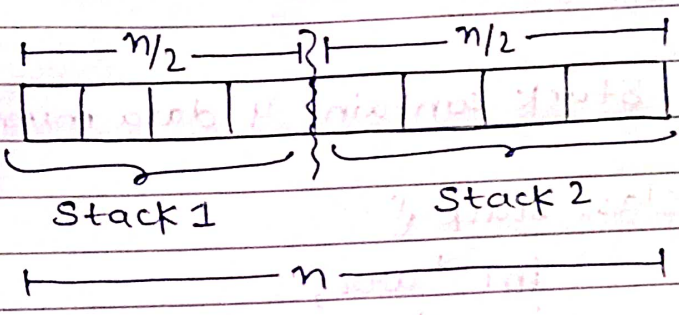
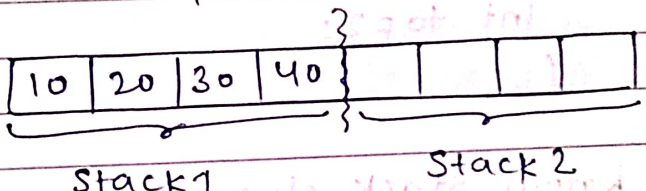
10/11/2023
Friday

* Implement 2 stack in a single array -

Approach 1 -



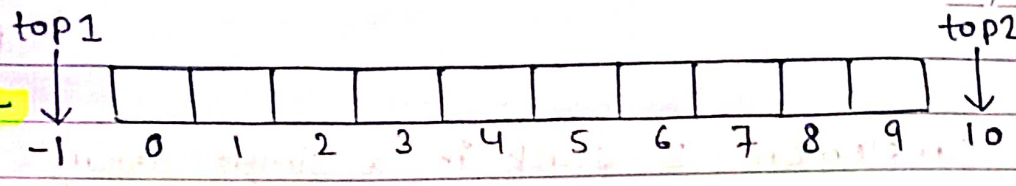
let we have push the elements in stack 1 -



Now, we have to push one more element 50 in stack 1 but stack 1 is full and stack 2 is empty. So, we can't insert more elements in stack 1.

See, the approach is not good to divide the array into two equal halves as lot of memory gets wasted.
But it is the one of the approaches.

Approach2 -



≈ class Stack contains 4 data members -

```
class Stack {
    int *arr;
    int size;
    int top1;
    int top2;
}
```

≈ As we know, stack elements can be accessed only from top. Let →

- Stack 1 has top1.
- Stack 2 has top2.

≈ Initially, top1 is at -1.
top2 is at size. (Here, size is size of array)

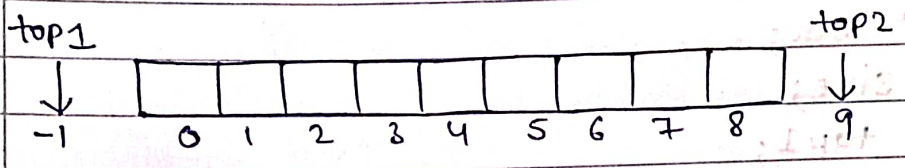
≈ **Conditions of underflow -**

- When, top1 == -1, Stack 1 is empty.
- When top2 == size, stack 2 is empty.

≈ **Conditions of overflow -**

- Jb top1 and top2 adjacent positions pr ho, it means there is no more space in the stack.

i.e. top2 - top1 == 1



Push –

in stack 1 → push1 (int data){
top1++;
arr[top1] = data;
}

in stack 2 → push2 (int data) {
top2--;
arr[top2] = data;
}

Pop –

from stack 1 → pop1 ( ) {
top1--;
}

from stack 2 → pop2 ( ) {
top2++;
}

Note – Remember, always apply the condition of underflow and overflow.

Code for implementing 2 stack in single array -

```
class Stack {
public:
int * arr;
int size;
int top1;
int top2;

Stack (int size) {
    arr = new int [size];
    this -> size = size;
    this -> top1 = -1;
    this -> top2 = size;
}

void push1 (int data) {
    if (top2 - top1 == 1) {
        cout << "overflow" << endl;
    }
    else {
        top1++;
        arr[top1] = data;
    }
}

void push2 (int data) {
    if (top2 - top1 == 1) {
        cout << "overflow" << endl;
    }
    else {
        top2--;
        arr [top2] = data;
    } }
```

```
void pop1 (){
    if (top1 == -1){
        cout<<"underflow"<<endl;
    }
    else{
        top1--;
    }
}

void pop2 (){
    if (top2 == size){
        cout<<"underflow"<<endl;
    }
    else{
        top2++;
    }
}
```

* **Valid parentheses –**

Given a string s containining just characters '(', ')', '[', ']', '{', '}'. Determine if the input string is valid.

An input string is valid if –

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of same type.

Example 1 –

I/P – s = "()"
O/P – true

Example 2 – I/P – s = "{[]()}"
O/P – true

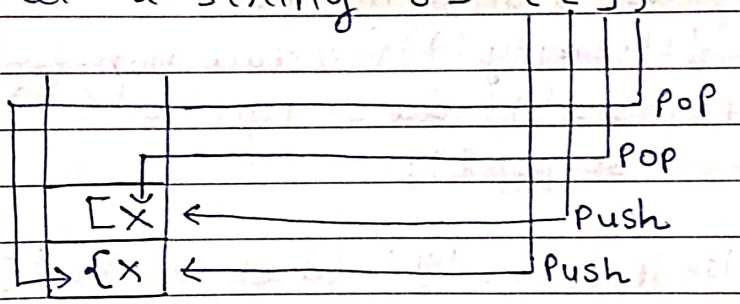
Example 3 –

I/P – s = "([)]"
O/P – false

Approach -

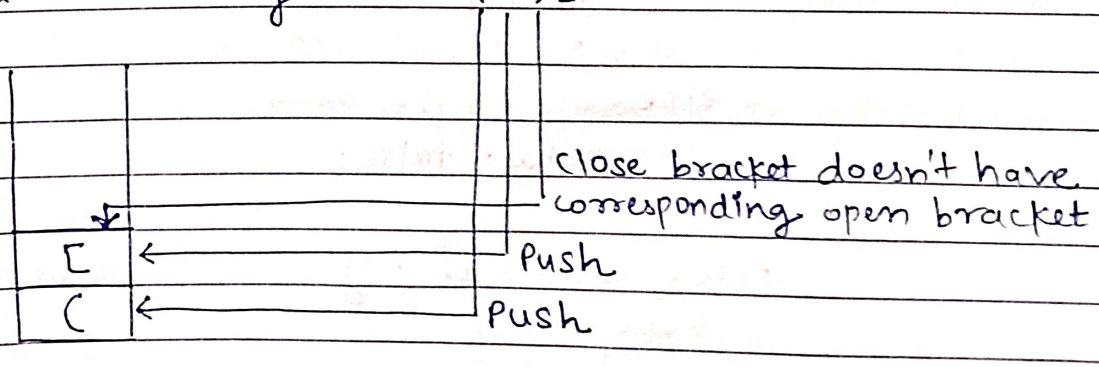
- Make a stack of characters.
- When encounter an open bracket in string, push it to the top of stack.
- When encounter a close bracket in string, check if the top of stack has corresponding open bracket of same type. If yes, pop it from the stack, otherwise return false.
- After traversing whole string, if stack is empty, it means valid parentheses otherwise invalid.

- Let a string s = "{[]}"



Now, stack is empty. It means valid parentheses.

- Let a string s = "([)]"



So, invalid parentheses.

Code –

```
class solution {
public:
    bool isValid(string s){
        stack<char>st;
        for(int i=0; i<s.length(); i++){
            char ch= s[i];
            if(ch=='(' || ch=='[' || ch=='{'){
                //open bracket mili hai
                st.push(ch);
            }
            else{
                //close bracket mili hai
                if(!st.empty()){ //stack non-empty
                    if(ch==')' && st.top()=='('){
                        st.pop();
                    }
                    else if(ch==']' && st.top()=='['){
                        st.pop();
                    }
                    else if(ch=='}' && st.top()=='{'){
                        st.pop();
                    }
                    else{
                        return false;
                    }
                }
                else{ //stack empty
                    return false;
                }}}
        if(st.size()==0){
            return true;
        }
        else{
            return false;}}};
```

* Check Redundant Brackets -

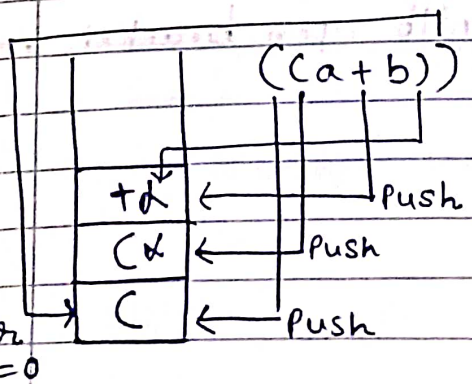
I/P - ((a+b) * (c+d))
O/P - Redundant brackets not present.

I/P - (a) * b
O/P - Redundant brackets present.

Approach -

- Make a stack of character.
- If character is open bracket or any operator (+, -, *, /) in that case push the character in stack.
- If character is close bracket, in that case count the operators until topmost character in stack is open bracket '(' and also pop out operator.
- Jb open bracket mil jaye at top then usko bhi pop out krdo.
- Now, if operatorCount is 0, return true.
- If saari string traverse kr li and also operatorCount > 0, in that case return false.

* Let the input string s = "((a+b))"



Redundant brackets present.

Code -

```
#include <bits/stdc++.h>
using namespace std;

bool checkRedundant (string &s){

    stack<char>st;
    for(int i=0; i<s.length(); i++){
        char ch= s[i];
        if(ch=='(' || ch=='+' || ch=='-' || ch=='*' ||
            ch=='/'){
            st.push(ch);
        }
        else{
            //close bracket found
            if(ch==')'){
                int operatorCount=0;
                while(!st.empty() && st.top()!='('){
                    char temp= st.top();
                    if(temp=='+' || temp=='-' || temp=='*' ||
                        temp=='/'){
                            operatorCount++;
                        }
                    // operator ko pop krdo
                    st.pop();
                }
                //yha pr pahuche iska mtlb open bracket mil
                //gyi at top
                st.pop();
```

```
if (operatorCount == 0){
        return true;
    }
  }
 }
}
return false;
}
int main(){
    string s = "a+(b)+c";
    bool ans = checkRedundant(s);
    if(ans == true){
        cout << "redundant brackets present";
    }
    else{
        cout << "redundant brackets not
                present" << endl;
    }
}
```

15/11/2023
Wednesday

* Min Stack (leetcode - 155)

Design a stack that supports, push, pop, top and retrieving the minimum element in constant time.

Example 1 -

I/P - ["MinStack","Push","Push","Push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

O/P - [null, null, null, null, -3, null, 0, -2]

Explanation -

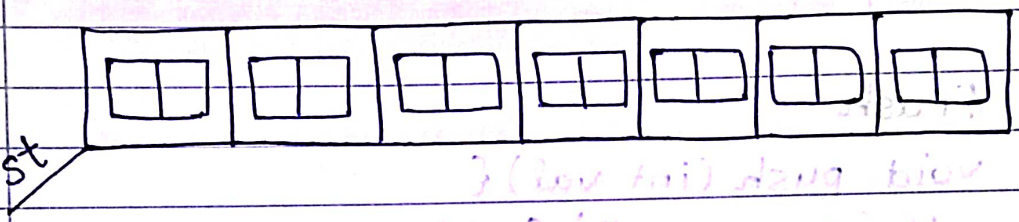
```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.Push(0);
minStack.push(-3);
minStack.getMin(); //return -3
minstack.pop();
minStack.top(); //return 0
minStack.getMin(); //return -2
```

Constraint - Method pop, top, getMin operation will always be called on non-empty stacks.

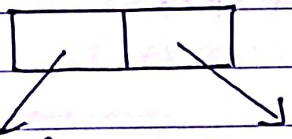
→ Normally, stack takes O(n) time to retrieve minimum element.

→ Using min stack, we can do this in O(1) time.

Let a vector st →



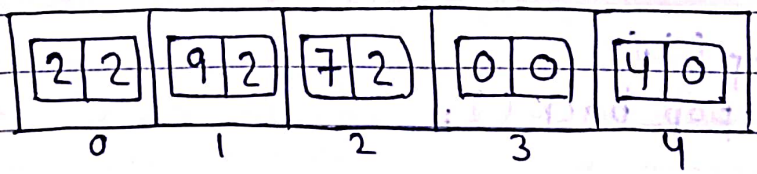
Har ek block mai 2 types of data stored hai.



Actual value or element

Current index tak ka minimum element

Every block of vector contains these two data.



So, in the vector [2,9,7,0,4], 0 is the minimum element.

Code-

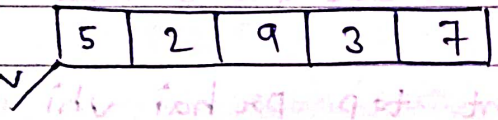
```
class MinStack {
public:

vector <pair <int, int > > st;

  MinStack ( ) {

    }

  //Push
  void push (int val) {
    if (st.empty()) { //in case vector is empty
      pair<int, int > p = make_pair (val, val);
      st.push_back (p);
    }
    else {
      pair <int, int> p;
      p.first = val;
      p.second = min (val, st.back().second);
      st.push_back (p);
    }
  }
  // pop
  void pop ( ){
      st.pop_back ( );
    }
  // top
  voi int top ( ) {
        return st.back ( ).first;
      }
  // minimum element
    int getMin ( ) {
        return st.back ( ).second; } };
```

* **next smaller element –**

→ If next smaller element present hai, then vo push krdo vector mai, otherwise -1 krdo.

I/P – v = [5, 2, 9, 3, 7]

O/P – [2, -1, 3, -1, -1]

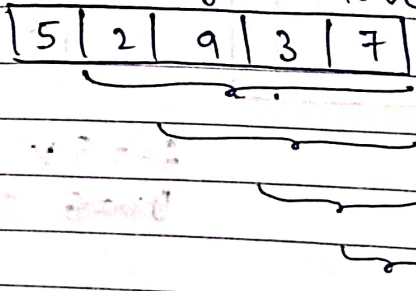


→ After 5, smallest element than 5 is 2.
→ After 2, no element is smaller than 2, so -1.
→ After 9, smaller element than 9 is 3.
→ After 3, no element is smaller than 2, so -1.
→ After 7, no element is smaller than 7, so -1.

So, O/P – | 2 | -1 | 3 | -1 | -1 |

Brute force Approach –

* hr i^{th} element ke liye $[(i+1)^{th}$ to $(size-1)^{th}]$ range mai next smaller element search kro.
  - Isme left to right move kr rhe hai.



i.e $n[(n-1)+(n-2)+(n-3)+\dots+1+0] = O(n^2)$

optimized Approach using stack →

__/__/____

Algo -

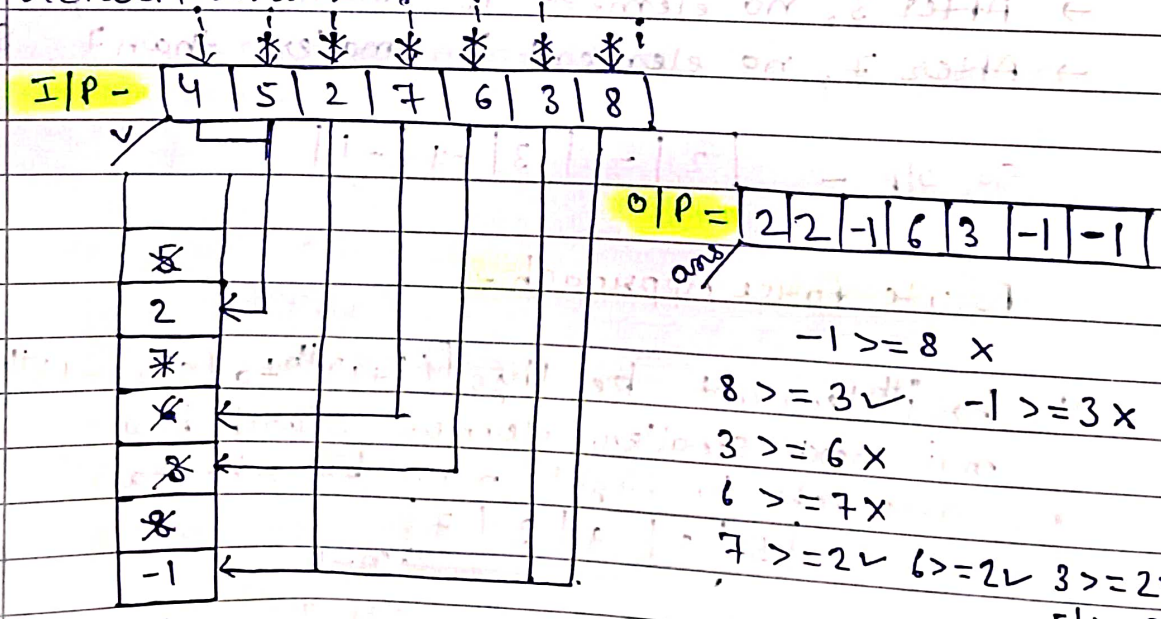
(1) Create ans vector which will store next smaller element corresponding to current element.

(2) Create a stack st.

(3) Initially push (-1) in stack. Here, we are moving right to left in given vector. As we know, last element of vector ka ans always -1 hoga kyuki uske next mai koi element hi nhi hai.

(4) Traverse the vector from right to the left.
Inside that loop curr = v[i].
Jb tak st.top() >= curr, tab tak st.pop() kro.
Now, jo bhi element top pr hai, vhi ans hai.
So, ans[i] = st.top();
Every time, push the curr element in stack because vo bhi kisi ka ans ho skta hai.

(5) return ans.



Code -

```
vector <int> nextSmallerElement (vector <int> &v)
{
    vector <int> ans(v.size());
    stack<int> st;
    st.push(-1);

    for (int i = v.size(); i>=0; i--) {
        int curr = v[i];
        while (st.top() >= curr) {
            st.pop();
        }
        ans[i] = st.top();
        st.push(curr);
    }

    return ans;
}
```

Time Complexity - O(n).

Note - Yha pr stack ka use track rkhne ke liye kiya hai elements ka.
stack har element ke corresponding jo ans aayega uska track rkh rha hai.

* prev smaller element -

→ If prev smaller element present hai, then vo push krdo, otherwise -1 krdo.

I/P - v = [8, 4, 6, 2, 3]

O/P - [-1, -1, 4, -1, 2]

8	4	6	2	3

→ Before 8, no element is smaller than it. So -1.
→ Before 4, no element is smaller than it. So -1.
→ Before 6, smaller element than it is 4.
→ Before 2, smaller element doesn't exists. So, -1.
→ Before 3, smaller element than it is 2.

So, O/P = | -1 | -1 | 4 | -1 | 2 |

Brute force Approach -

Right to left previous smaller element search krenge.
T.C. → $O(n^2)$

Optimized approach using stack -

Same as finding next smaller element. The difference here is left to right move krenge isme.

Code -

```
vector<int> prevSmallerElement(vector<int> &v)
{
    vector<int> ans(v.size());
    stack<int> st;
    st.push(-1);

    for(int i=0; i<v.size(); i++){
        int curr = v[i];
        while(st.top() >= curr){
            st.pop();
        }
        ans[i] = st.top();
        st.push(curr);
    }

    return ans;
}
```

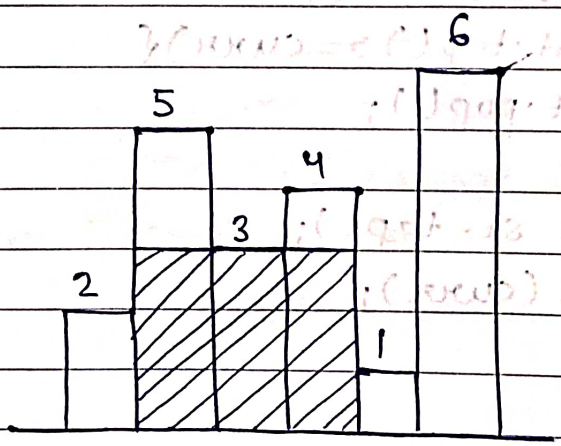
* Largest Rectangle in Histogram (leetcode - 84)

Given an array of integers heights, representing the histogram's bar height where the width of each bar is 1, return the area of the largest rectangle in the histogram.

Example 1 -

I/P - heights = [2, 5, 3, 4, 1, 6]

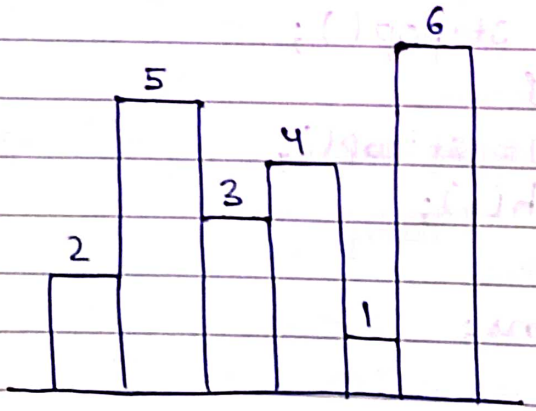
O/P - 9



The shaded region is the largest rectangle in histogram.

Approach -

(1) Find indices of next smaller elements in array heights.
(2) if (next[i] == -1) next[i] = heights.size()
kyuki next smaller element index can't be -1.
(3) Find indices of previous smaller elements in array heights.
(4) Traverse a loop on array heights.
(5) width = next[i] - prev[i] - 1, length = heights[i].
(6) area = length * width.
(7) Return maximum area of all.



next

4	2	4	4	6	6

prev

-1	0	0	2	-1	4

width =	4	1	3	1	6	1
length =	2	5	3	4	1	6

area =	8	5	9	4	6	6

→ maxArea

Code -

```
class solution{
public:
  vector<int>nextSmallerElements(vector<int> &v)
  {
    vector<int>ans(v.size());
    stack<int>st;
    st.push(-1);
    for(int i=0;

    for(int i= v.size()-1; i>=0; i--){
      int curr = v[i];
      while(st.top() != -1 && v[st.top()]>=curr){
        st.pop();
      }
      ans[i]= st.top();
      st.push(i);
    }
  return ans;
}

vector<int>prevSmallerElements(vector<int>&v){
    vector<int>ans(v.size());
    stack<int>st;
    st.push(-1);
    for(int i=0; i<v.size(); i++){
      int curr= v[i];
      while(st.top()!=-1 && v[st.top()]>=curr){
        st.pop();
      }
    ans[i]= st.top();
    st.push(i);
  }
return ans; }
```

```
int largestRectangleArea (vector <int> &heights){

    vector <int> next = nextSmallerElements (heights);

    for (int i=0; i < heights.size(); i++) {
        if (next[i] == -1 ) {
            next[i] = heights.size();
        }
    }

    vector <int> prev = prevSmallerElements (heights);
    int area = 0;
    int maxArea = INT_MIN;

    for (int i=0; i< heights.size(); i++ ) {
        int width = next[i] - prev[i] - 1;
        int length = heights[i];
        area = length * width;
        maxArea = max (maxArea, area);
    }

    return maxArea;
  }
};
```